Research report 78

ARCA – A NOTATION FOR DISPLAYING AND MANIPULATING COMBINATORIAL DIAGRAMS

by

Meurig Beynon

(RR78)

Abstract

ARCA is a programming notation which was originally developed with the computer animation of a class of diagrams studied by ARthur CAyley in mind. It is presented here as an archetypal example of a "definitive notation", in a sense explained in a previous paper by the author.

This paper is an informal introduction to the principles of ARCA and the ARCA system, and includes some tutorial examples.

Department of Computer Science
University of Warwick
Coventry, CV4 7AL, England

June 1986

ARCA - a notation for displaying and manipulating combinatorial diagrams

Meurig Beynon.

Department of Computer Science, University of Warwick.

ABSTRACT

ARCA is a programming notation which was originally developed with the computer animation of a class of diagrams studied by ARthur CAyley in mind. It is presented here as an archetypal example of a "definitive notation", in a sense explained in a previous paper by the author.

This paper is an informal introduction to the principles of ARCA and the ARCA system, and includes some tutorial examples.

Introduction.

In many contexts, geometrical diagrams have proved to be a powerful means of representing information. Their importance in geometric design and modelling is self-evident, but they can also be helpful in describing abstract concepts whose geometric nature is less apparent. Combinatorial graphs - comprising vertices (possibly labelled), together with edges (possibly directed, labelled or coloured) - provide many examples of diagrams of the latter kind: circuit diagrams, transition diagrams for finite state machines, and Hasse diagrams for partially ordered sets and lattices. Typically, the semantic content of such a graph is implicit in the incidence relations, but can be inspected only when an appropriate geometric realisation is constructed. Appropriate computer-aided design tools can be used both to construct such realisations, and to assist subsequent interaction and interpretation.

ARCA is a programming notation which was originally designed with the interactive display and manipulation of "Cayley diagrams" - a particular class of combinatorial graphs - in mind. The chief characteristics of Cayley diagrams (CDs), and the special problems posed by their representation are outlined in §1 and §2 below. Though ARCA was designed for a specific application, the ARCA system seems likely to prove useful for a variety of purposes, and features of the design itself may have wider interest. In particular, in a sense explained in [2], ARCA is here presented as an archetypal (or even ARCAtypal!) example of "a definitive notation".

This paper sketches the principles which underlie the design of ARCA informally; for fuller details the interested reader should consult [1] and [2].

§1. Cayley diagrams and graphs.

Cayley diagrams were devised by Arthur Cayley (1821-1895) for representing group-theoretic relations pictorially. Simple examples of Cayley diagrams (CD) appear in Fig's.1-4. (In interpreting these diagrams, all oriented edges are to be one colour [red], and all unoriented edges are to be bidirected edges of another colour [green].) It will be helpful to distinguish between a CD and the abstract graph - a "Cayley graph" (CG) - which it depicts. A brief sketch of some of the principal features of CD's (as they relate to the design of ARCA) is given here; for further details, see [4]. In the sequel, a superficial understanding of the relationship between groups and CD's will suffice.

The relationship between a CG and its associated group is most easily illustrated with reference to Fig.1. It is a simple exercise to the reader to label the vertices of Fig.1. with the six permutations of the set {1,2,3} in such a way that the vertex with label p is connected to the vertex with label p.(1,2,3) under a red edge, and the vertex with label p.(1,2) under a green edge. If R and G respectively denote the permutations (1,2,3) and (1,2), it is then trivial to compute any product of R's and G's by tracing a path of appropriately coloured edges in Fig.1.

It is easy to verify that each of the three products

$$R^3 \ , \ G^2 \text{ and } (R.G)^2$$

represents the identity permutation. Indeed all *relations* between R and G (ie product of R's and G's defining the identity) can be derived from the 3 relations above simply by using the group axioms. For instance:

$$R.G.R^2.G.R = R.G.R^2.G.R.G^2 = R.G.R.(R.G)^2.G = (R.G)^2 = 1,$$

since $G^2 = 1$, and multiplication is associative.

Formally, the CG of Fig.1. is associated with the group freely generated by two generators r and g subject to the relations:

$$r^3 = g^2 = (r.g)^2 = 1,$$

or equivalently, with the *presentation:*

$$< r, g \mid r^3 = g^2 = (r.g)^2 = 1 > \qquad (1.1)$$

of the symmetric group S_3.

In general, a CG is specified either by exhibiting a set of generators for a concretely presented finite group (e.g. {R,G} for S_3), or by giving a particular presentation for an abstract finite group (e.g. the presentation (1.1) for S_3). All vertices of a CG are equivalent under symmetry, and if one is selected as the initial and unique final state the resulting finite state machine recognises relations over the alphabet of generators. This interpretation of a CG is useful when constructing a CD, since it provides an intrinsic method of referencing one vertex relative to another by specifying an appropriate path via a string of generators (or inverses of generators, which correspond to traversing directed edges in the opposite sense).

The group-theoretic information in a CD is captured in the incidence relations between vertices and edges, which can be derived from an abstract group presentation (such as (1.1)) by the well-known technique of "coset enumeration". The specification of coordinates for the vertices of a CD has no purpose other than to aid the apprehension of the relations between the generators, and to provide a visual image of the group. The problem of realising an abstract CD effectively resembles that of laying out an abstract circuit, in that the relevent criteria are aesthetic and pragmatic, and are not easy to specify precisely. It is obviously desirable that a planar realisation should preserve symmetry as far as possible, that paths should be easily traceable, and that edge-crossings should be introduced only if they enhance symmetry or assist readability. Some pioneer work on the realisation of CDs was carried out by Maschke [6], who classified and constructed realisations of CDs which can be embedded in the plane without crossings. Fig.4 is a "good" realisation (due to Maschke) of the CD associated with the presentation:

$$< x,y \mid x^2 = y^5 = (xy)^3 = 1 >$$

of the alternating group A_5.

The problem of constructing a good 2-dimensional realisation of a CG is clearly closely connected with the underlying group structure, but has no simple general solution. Though coset enumeration can determine the incidences of a CG, it generates a haphazard indexing of vertices which does not aid its realisation as a CD. In special cases, there are group-theoretic methods of constructing symmetric realisations of CGs, but these are generally in Euclidean spaces of dimension higher than 2. For instance, there is a natural way to construct higher dimensional CDs for a direct or semi-direct product from CDs representing its components. However, even if it were possible to devise effective methods of constructing CDs automatically, a notation such as ARCA would still be very useful, as explained in §2.

§2. The background to ARCA.

Generalities.

In many applications, it is important to recognise that a visual image is of limited use when divorced from the underlying conceptual model. In devising a graphics system for such an application, it is not enough merely to provide for efficient display; it must be possible to specify the conceptual models underlying images simply and systematically. This is the case for large CDs, where generation of a picture is of little value without a medium for referencing vertices and edges for group-theoretic purposes. It is helpful, for instance, to display paths of edges defining relations, products of group elements, or subgroups. It is also interesting to examine the consequences of introducing a new generator (which is graphically equivalent to replacing each instance of a particular sequence of directed and coloured edges by a single edge of a new colour), or forming a quotient group (which entails identifying vertices lying in the same coset of a normal subgroup).

ARCA is conceived as a medium to be used (possibly in conjunction with automated techniques) for constructing computer representations of CDs to assist comprehension, display and manipulation. A CD has considerable semantic content, and the data structure needed to represent it is correspondingly complex. In view of this, the exclusive use of a graphics interface is inappropriate, and the ARCA system has two interfaces with the user: a primary screen on which the text of an ARCA program is developed, and an auxiliary screen for graphical display as and when required.

ARCA as a "definitive notation".

ARCA has been designed as a "definitive notation" in a sense explained fully in [2]. In developing a definitive notation for a particular application, an appropriate underlying algebra of data types and operations must first be chosen.

When displaying and manipulating CDs, it is necessary to specify scalar, vector and incidence information. In ARCA such information is respectively represented by integers (of specified modulus), vectors of integers (of specified dimension), and perms i.e. permutations / partially defined permutations (of specified degree). The data types in the underlying algebra comprise these three *primitive* types, together with a complex data type which is used to represent (partial) CDs. There are numerous algebraic operators which relate integers, vectors and perms (e.g. arithmetic operations on integers, vector operations such as addition, rotation and reflection, scalar product, and composition and superposition of perms), which together define the *primitive data algebra*. There are also special operators (such as are needed to join subgraphs, or index such a join).

Following the principles described in [2], a definitive notation includes variables which denote implicitly or explicitly defined values in the underlying algebra. The values of variables are determined by a sequence of *definitions*, each of which *either* assigns a formula *or* a specific value to a variable. Semantically, a formula assignment:

$$a = f(b,c,...,z),$$

where f is a formula over the underlying algebra in the variables b,c,...,z , asserts that (until redefinition occurs) the value of the variable a is to be determined as and when required by evaluating the formula f(b,c,...,z) over the underlying algebra. The value of the variable a is then implicitly defined in terms of the values of other variables. In the particular case where the formula f is constant, such a definition specifies an explicit value for the variable a.

In the above context, g may be used to denote the value of a subexpression g. Note that definitions which lead to circularity, such as:

$$a=b;\ b=c;\ c=a+2$$

are trapped as semantic errors, but that a definition such as

$$a = |a+c|+b$$

which defines a by the formula

'b+α, where α is the value of (a+c) current at the point of definition'

is permissible.

A definitive notation is limited as a "declarative" notation by the absence of recursive definition, and as a "procedural" notation by the absence of user-defined procedures. The natural way to seek to compensate for this is to allow the user to enhance the basic data algebra by defining new operators and/or new data types. Without clear abstract principles to determine e.g. whether assignment to the components of a variable of a new hierarchical data type should be permitted, and how such variables should be declared, the problem of extending the data types is difficult (c.f. [2]). Making provision for user-defined operators is more straightforward, and this feature is incorporated in ARCA (c.f. §3, Example 3).

Representing CDs in ARCA

The computer representation of a CD has to capture two aspects: the group-theoretic interpretation as a CG, and the geometry of the diagram. In ARCA, scalars, vectors, and perms are represented by 'primitive' variables of type *integer*, *vertex* and *colour* respectively. The type of a variable is specified on declaration, and type of any algebraic expression can be determined statically. To allow convenient incremental specification of *vertex* and *colour* values, as well as implicit definition of an entire *vertex* or *colour* by a single formula of the appropriate type, there are two kinds of *vertex* and *colour* variables: *composite* and *abstract*. A *composite vertex* (resp. *colour*) variable has a fixed dimension (resp. degree) specified on declaration, and is semantically equivalent to an array of *integer* variables. Such variables must be defined componentwise by a family of formulae of *integer* type. By contrast, an *abstract vertex* (resp. *colour*) variable must be defined by means of a formula of type *vertex* (resp. *colour*). (For a fuller discussion, see [2].)

Note that an ARCA *vertex* variable, whether *composite* or *abstract*, represents an "abstract graph

vertex" rather than a geometrical point. There is no way of "equivalencing" a pair of *vertex* variables, and distinct vertices, whether declared within a diagram or independently, serve as distinct names for abstract vertices. By making appropriate assignments, it is nonetheless easy to ensure that two *vertex* variables represent the same geometrical point. For instance, if u,v and w are declared *abstract*, the assignments

u:=w; v:=w

will ensure that the coordinates of u and v coincide with those of w until such time as new formulae are assigned to u or v.

ARCA *vertex* variables provide a means of referencing vertices of an abstract graph which is essential for satisfactory representation of a CG. As explained above, it is easy to generate the incidences of a CG without having any coordinates for the vertices in mind, which makes direct means of reference (e.g. via a mouse or light-pen) of limited use. In any case, even an array which supplies an index for each abstract vertex of a CG may be unhelpful in group-theoretic terms. A satisfactory representation of a CG requires an array of indices for abstract vertices together with the transition (a permutation of indices) induced by each generator. In an ARCA program, this information is associated with a variable of type *diagram*, as explained more fully in §3. The *diagram* syntax is designed to assist referencing of vertices relative to each other via paths of edges (see §1). This mode of referencing can be used iteratively to specify subgroups or subsets of vertices with special symmetries.

In describing the geometry of a CD, it is often necessary to specify geometrical relationships between vertices. When seeking a good realisation of a particular CG, it may be helpful (for instance) to constrain a subset of vertices to define a square, or to be collinear. The vertex array of an ARCA *diagram* can consist of *abstract* or *composite* variables (depending on the mode of declaration), so that the coordinates of vertices of diagrams can be constrained to satisfy vector relationships or scalar relationships between components by making appropriate definitions.

Implicit definition can also be useful for specifying incidence information. For instance, it is often possible to transform one CD to another by changing the orientation of a suitable subset of edges. By defining incidences implicitly in such a case, a single ARCA *diagram* can be used to represent two or more CDs (see §3 Example 2).

Comparison with SKETCHPAD.

The main principles which are used in ARCA to describe representations of CDs have been outlined above. The problems of displaying and manipulating combinatorial graphs are of course relevant to most graphics systems, and a brief comparison with the classical approach described in SKETCHPAD [7] may be of interest.

In this comparison, it is significant that SKETCHPAD, in common with many systems which use a single graphics interface, combines two objectives:

(a) the interactive specification of computer representations of a visual image and its conceptual model

(b) the use of a graphical as opposed to a conventional textual interface for communication.

The practical advantages of (b) are clear: direct reference to the elements of the visual image, freedom from the problems of coordinatisation, and the use of analogue methods for manipulating the image. The chief limitation of (b) is that the conceptual model is not necessarily easily accessible via the visual image, and that the semantic content of such a model is often too rich to be described without a formal notation. Graphical systems which emphasise (b) may also be difficult to describe formally, and are best appreciated through practical experience.

The case for the separation of concerns (a) and (b) (c.f. [3] p.211) is obvious, and accounts for the emphasis on (a) in ARCA. Though an ARCA program may superficially resemble a source from which to compile graphical output, it should be stressed that ARCA is by design a notation for interactive use. For instance, the primary purpose of implicitly defined values is to allow geometrical relationships to be partially specified pending complete specification by inspection or experiment. In a practical ARCA system, it would be desirable to use the graphics interface less passively to relieve the heavy textual bias.

The counterpoint between declarative features (constraints) and procedural features (e.g. means of updating coordinate and incidence information) is a common characteristic of ARCA and SKETCHPAD. The use of functional rather than equational constraints in ARCA is a limitation which appears to have

certain advantages where dialogue is concerned. (For a fuller discussion of this point, see [2], where the merits of definitive notations for interaction, and the case for regarding constraint-based programming as adapted for "problem-solving" rather than "dialogue" is considered.)

In SKETCHPAD, the concept of "points and lines subject to constraints" is faithful to geometric intuition. Points and lines are visible geometric entities, and "constraining four points to lie at the vertices of a square" genuinely defines an equational constraint upon four points. In ARCA, constraints apply to conceptual rather than geometric points, but are restricted in that they are functionally rather than equationally specified. For instance, four points will lie at the vertices of a square because the coordinates of three of the points are implicitly defined in terms of those of the fourth, but this constraint may be abstract in that no coordinates for the latter have been specified. Such an approach has the advantage that two conceptually distinct but geometrically coincident points may be subject to independent constraints.

§3. Programming in ARCA.

Some simple program fragments illustrating the main principles of ARCA are presented informally below. The commentaries on the programs which follow serve as a tutorial introduction to ARCA. For formal details, see [1].

The most direct way of representing a CD in ARCA is to define a *diagram* which describes the necessary incidence and coordinate information explicitly. In such a *diagram*, coordinates and incidences are respectively described by a *vertex array* and a *colour list*. Example 1 is an ARCA program which defines a *diagram* D to represent the CD depicted in Fig.1. The *diagram* D is declared at line 3, and the colours and vertices of D are defined at line 4-5 and 6-9 respectively.

Example 1.

```
1     vert 2: v;
2     v = [0,1];
3     'ab'-diag (vert 2, col 6) : D;
4     a_ D = {1,3,5%6}{2,6,4%6};
5     b_ D = {1,2%6}{3,4%6}\{5,6%6};
6     D!1 = [0,1%1]; D!2 = [0,2%1];
7     with i in 3 : I = 2,3 do
8           D!(2*I) = rot(D!2,I-1,v);
9           D!(2*I-1) = rot(D!1,I-1,v)
10    od
```

Declaration of a *vertex* variable is illustrated at line 1. By default (i.e. unless the keyword *abst* is used), primitive variables are taken to be *composite* on declaration. A primitive variable may have an associated non-negative integer *weight*: this is the *modulus* of an *integer* variable, the *dimension* of a *vertex* variable, and the *degree* of a *colour* variable. The *weight* of a variable gives information about the values it can represent, and must be specified when an *composite* variable is declared. As an example, the variable v declared at line 1 has *dimension* 2, and is used to represent coordinates in the plane. A *colour* variable of *degree* r represents a perm of the set { 1 , 2 , . . . , r } of residues mod r. An *integer* variable of *modulus* d represents a residue mod d if d ≥ 2, and a traditional integer if d=0. An "*integer* of *modulus* 1" has a special interpretation, and - in a sense explained in [1] - "1 modulo 1" represents "a suitable geometric unit for purposes of display."

Many of the standard operators in ARCA are intended to simplify the specification and manipulation of constants of primitive type. The operator "%" is used when specifying integer constants modulo a base, so that "8%3" and "2%3" both represent the residue 2 modulo 3 (c.f. line 6). Cross-modulus arithmetic is illegal in general, but an *integer* of *modulus* 0 may be coerced to a particular modulus in context. The postfix operator " ' ", which returns the *principal value* - the unique representative in the range $1 \le r' \le n$ of a residue r modulo n, assists translation between moduli (c.f. lines 8-9). Though 'cyclic notation' is an excellent way of denoting traditional permutations, denoting perms which are partial functions poses some problems. This is illustrated at line 4, where the RHS denotes the perm (1,3,5)(2,6,4) in conventional cycle notation. In ARCA, this perm is most conveniently described as the *superposition* (\) of {1,3,5} - the perm

which maps 1,3,5 cyclically but is otherwise undefined - and {2,6,4}. Note that the degree of a perm such as {1,3,5} is ambiguous, and suitable conventions for inferring *weight* information are needed if expressions such as {1%6,3%6,5%6} are to be avoided (c.f. [1] §3.3).

The *with*-loop is semantically similar to a conventional *for*-loop, and incorporates a specification of a special "control variable", which resembles an integer variable in a conventional procedural programming language. The *with*-loop is in effect equivalent to the four definitions obtained by substituting I=1 and I=2 into the definitions at lines 8 and 9. The operator rot(,,) is used at lines 8-9 to denote planar rotation. An *integer* of *modulus* 2 or more is required as the second parameter, and is interpreted as an angular measure; thus

rot(D!2,1%3,v)

represents "the vector obtained by rotating (the coordinate vector of) D!2 anti-clockwise through $\pi/3$ radians about [0,0]."

The *diagram* D declared at line 3 has two colours, denoted a_ D and b_ D, and six vertices, denoted D!1, ..., D!6. The vertices and colours are specified (by default) as *composite* (c.f. Example 2 line 3), and have *dimension* 2 and *degree* 6 respectively. Note that all vertices of D have the same *dimension*, and all colours of D have the same *degree*, which is necessarily also the number of vertices in D.

In Example 1, all definitions specify explicit values for variables. The operators ([ ], { }, %, \, *, +, rot() etc.) which appear on the RHS's of these definitions all belong to the primitive data algebra (c.f. §2 and [1] §2). By replacing line 3 of Example 1 by

3 'ab'-diag (abst vert , col 6) : D;

and deleting line 6, the definitions at line 8-9 specify the coordinates of the vertices D!3, ...,D!6 implicitly in terms of D!1 and D!2, and a *diagram* to represent a family of realisations of the CG of Fig.1 is obtained. In each realisation, the triples (D!1,D!3,D!5) and (D!2,D!4,D!6) are at the vertices of equilateral triangles centred on [0,0], but D!1 and D!2 are at points which can be independently specified. Thus the geometric configuration could be completely specified by "D!1 = [0,3%1] ; D!2 = [0,4%1]" *or* "D!2 = 2.D!1 ; D!1 = [0,2%1]". In the latter case, the coordinates of all the vertices of D would depend on the coordinates of D!1.

The CD of Fig.1 is closely related to the CD for the presentation $< x , y \mid x^2 = y^3 = 1$ and $xy = yx >$ of the Abelian group $C_2 \times C_3$ depicted in Fig.2. The geometrical relationship between the two reflects a group-theoretic relationship; the group D_3 is a 'semi-direct' product of C_2 and C_3 (see [5] p.88-90). Example 2 below generates a *diagram* D which represents the abstract graph of Fig.1 or Fig.2 according to whether the value of the *integer* variable i is 0 or 1, and defines different planar realisations subject to the current values of D!1 and D!2. For this purpose, both the vertices and colours of D are specified as *abstract* at line 3. The "@" operator at line 5 denotes exponentiation of perms, and takes precedence over superposition.

Example 2.

```
1   vert 2: v;
2   v=[0,0];
3   'ab'-diag (abst vert 2, abst col 6) : D;
4   int : i;
5   a_ D = {1,3,5%6}\{2,4,6%6}@(1-2*i);
6   b_ D = {1,2%6}\{3,4%6}\{5,6%6};
7   with int 3 : I = 2,3 do
8           D!(2*I') = rot(D!2,I-1,v);
9           D!(2*I'-1) = rot(D!1,I-1,v)
10  od
```

Examples 1 and 2 illustrate the use of *composite diagram* variables. To allow more powerful methods of manipulating diagrams, ARCA includes operators which act on diagrams, and *abstract diagram* variables. A *composite diagram* variable can only represent graphs of fixed size, unlike an *abstract diagram* variable, to which "a formula defining a *diagram* implicitly" as opposed to "a family of formulae defining the components of a *diagram* implicitly" can be assigned (see [1] §5). This is illustrated

in [1] §9, where an *abstract diagram* to represent a generic class of CDs including Fig.'s 1-3 is defined.

Example 3 below is an ARCA program defining a *diagram* T to represent the CD depicted in Fig.4, and illustrates most ARCA features other than the *abstract diagram*. Lines 9-16 illustrate the definition of an operator (see [1], §8); $1,$2 and $3 being formal parameters. Clauses of the form "// ... " are comments.

The required *diagram* ('ab'-T) is constructed by first defining the skeletal subdiagram 'a'-T (lines 17-26), then inserting the edges of colour 'b'. The *diagram* 'a'-T is synthesised from four components: the innermost and outermost pentagons ('a'-P and 'a'-S), and the two "pentagons of pentagons" ('a'-Q and 'a'-R) which enclose 'a'-P and are enclosed in 'a'-S. The subdiagrams 'a'-Q and 'a'-R are defined using *diagram* product (i.e. direct product of graphs) at lines 19-20.

The operator CS is used to define regular pentagons, appropriately scaled, oriented and centred. Assignment to a *composite diagram* is used at lines 17-18 to specify the subdiagrams 'a'-P and 'a'-S, and at lines 23-24 to specify the coordinates of Q and R, where "Q/<m..n>" denotes the restriction of the *diagram* Q to the set of vertices with indices [|m|, |m+1|, ..., |n|]. (By convention, implicit indices used to reference components on the LHS of a definition are evaluated.) In all cases, coordinates of vertices are implicitly defined.

The skeleton of the final *diagram* ('a'-T), comprising 12 regular pentagons whose edges are of colour 'a', is defined as the join (i.e. disjoint union) of its four components at line 26. The coordinates of the vertices of T are defined implicitly by formulae depending on the *integer* r, which serves as a 'scaling parameter' for T.

The edges of colour 'b' incident with the innermost and outermost pentagons are defined in lines 27-33, using the indexing operator '**', where

$$i\%m ** j\%n \equiv (m(i'-1)+j') \% mn.$$

The use of colour paths for vertex referencing (c.f.§'s 1 and 2) is illustrated in lines 34-46. Note that the RHS of line 37 is an abbreviated form of

$$a\_\ T^{-}.b\_\ T.a\_\ T^{-},$$

where "." and "$^{-}$" respectively denote composition and inverse of perms.

Example 3.

```
1       abst int : r ;
2       int 60 : x , y ;
3       int 25 : m , n ;
4       vert 2 : z ;
5       z = [0 , 0];
6       'a'-diag (abst vert , col 5) : P , S ;
7       'a'-diag (abst vert , col 25) : Q , R ;
8       'ab'-diag (abst vert , col 60) : T;

9       op (vert 2 : $1 , $2 ; int 0 : $3) -> 'a'-diag (vert 2 , col 5) : C5 is
10              with int 5 : H = 1..5 do
11                      C5!H = rot ($2 , H-1 , $1);
12                      a_ C5{H} = H - 1 + 2 * $3
13              od
14      si ;
15      // the above operator returns a regular pentagon with centre $1,
16      // first vertex at $2, and orientation specified by $3 (= 1 or 0).

17      'a'-P = 'a'-C5 (z , [r,0] , 0);
18      'a'-S = 'a'-C5 (z , (-4).P!1 , 1) ;
19      'a'-Q = ''-P ** 'a'-P ;
20      'a'-R = ''-P ** 'a'-P ;
21      with int 5 : I = 1..5 do
22              m = I**1; n = I**5;
23              Q/<m..n> = C5 (2.P!I , (3/2).P!I , 0) ;
24              R/<m..n> = C5 ((-3).P!I , (-7/2).P!I , 0)
25      od ;

26      'a'-T = 'a'-P :: 'a'-Q :: 'a'-S :: 'a'-R ;

27      with int 5: J = 1..5 do
28              m = J ** 1 + 5 ;
29              with int 60: K = 0, 30 do
30                      b_ T{K+J'} = K+m' ;
31                      b_ T{K+m'} = K+J'
32              od
33      od ;

34      with int 60: K = 0, 30 do
35              with int 5: J = 1..5 do
36                      x = a.b_ T{K+J'} ;
37                      y = a¯.b.a_ T¯{K+J'} ;
38                      b_ T{x} = y ;
39                      b_ T{y} = x ;
40                      with col 60 : CC = |a_ T| , |a_ T¯| do
41                              x = CC@2.b_ T{K+J'} ;
42                              y = CC@2.b_ T.CC¯@2{K+30+J'} ;
43                              b_ T{x} = y
44                      od
45              od
46      od
```

§4. Implementing an ARCA system.

In this section the main features of the ARCA implementation currently under development are discussed. Although the creation of a sophisticated environment for developing ARCA programs is envisaged, the central point of reference for the design is the basic definitive notation as described in §2. The use of such a simple framework appears to have several merits, ensuring clear semantics whilst assisting modularity and extensibility.

The implementation of the definitive notation itself is conceptually simple, though there are some technical difficulties. The large number of operators in the underlying algebra makes it convenient to use a single token for distinct operators (e.g. '+' for both scalar and vector addition, '.' for multiplication of a vector by a scalar and perm product), and variable typing is used to allow syntactic disambiguation. Because of the central role of algebraic expressions, a definitive notation is well-adapted for the automatic construction of an LR-parser via a compiler-compiler. (See [1] for a YACC specification of the ARCA expression parser.)

The routines which are used in interpreting a definitive notation serve five main functions: compiling, simplifying, evaluating, tracing and displaying formulae. In the current implementation, the formula appearing on the RHS of a definition is compiled into a tree representation, simplified (e.g. by constant folding, or by evaluation of subexpressions where specified), traced to check for circularity, then associated with an abstract variable or composite variable component. Note that in the compilation phase it is possible to distinguish between formulae or subformulae which define explicit and implicit values. This is significant when a definition of one component of a composite variable in terms of a second component is to be interpreted. Thus if k is an *integer* variable, and v is an composite *vertex* variable, then a definition such as

$$v[1] = k.v[2]$$

is to be permitted, whilst the formally similar assignment

$$v[1] = 2.v[k]$$

must be deemed a semantic error, since it is potentially circular. This interpretation is achieved in the current implementation by "extracting explicitly indexed components of composite variables" in a suitable fashion during the simplification phase.

Simplification of formulae may serve several functions: it can be used for "optimising" defining formulae, which are in general frequently re-evaluated, or for the elimination of constructs which are primarily used for notational convenience (such as "1..5" for the list "1,2,3,4,5", or "a~.b@2.c_ D" for "a_ D~.b_ D@2.c_ D"). There is also scope for invoking axioms which apply to the underlying algebra, though this may not be appropriate for ARCA.

Evaluation of formulae is straightforward; it requires only a compendium of routines for evaluating primary operators in the underlying algebra. Undefined values can be handled gracefully in a definitive notation, and the evaluation routine should be adapted for this. For instance, when evaluating a composite *vertex* variable, the ARCA interpreter may return a vector value of the appropriate dimension in which some components are undefined. Efficient evaluation of standard operators can have a significant effect upon the efficiency of the entire implementation; for this reason, the primary evaluation routines merit "optimisation", and, in some applications, might justify the provision of special hardware. Efficiency can also be improved by storing the most recent evaluations of variables, and monitoring functional dependencies so as to avoid unnecessary re-evaluation.

In a suspended ARCA dialogue, the current context is determined by the existing definitions of variables. It is important that these definitions (which represent the "transient values" and "persistent relations" alluded to in [2]), should be available for inspection. To this end, the ARCA interpreter includes routines for reconstructing defining formulae from their compiled forms for display.

The above discussion deals with the implementation of the definitive notation upon which ARCA is based, and addresses the issues which are most central to the entire system. At present, the development of a suitable environment for exploiting ARCA fully is at an early stage, and it is only appropriate to outline some of the features which are required for effective use.

The primary need is for high-level commands to allow basic semantic actions (such as displaying a *diagram*, constructing the group table associated with a *diagram*, or highlighting the elements of a

subgroup within a displayed *diagram*) to be performed. Such actions have no side-effect upon the current state of the ARCA dialogue, and are for the most part easy to implement since the relevant data is conveniently stored and represented. There are some technical problems associated with the choice of coordinate system for *diagram* display; these are simply solved by ensuring that the components of vectors as specified by the user are conceptually "units of length" (represented in ARCA by "*integer*s of modulus 1"), and are scaled appropriately for purposes of display (cf [1]). In this context, the design of a satisfactory format for the group-theoretic and graphical "command languages" is more problematical than the implementation of commands. An interesting possible solution might be to interpret graphical and group-theoretic commands within the framework of auxiliary definitive notations.

The design of the interface for the ARCA system is another important concern. In essence, the process of defining and interrogating variables must be as convenient and transparent as possible. A method of synthesising new definitions by editting previous definitions (or sequences of definitions within a *with*-loop) might be useful. Another possibility might be use a mouse or light pen for determining the index of a vertex within a displayed *diagram*, or for re-defining the coordinates of a displayed *vertex*. As explained in §2, these methods could have limitations if several abstract vertices were represented by the same geometrical point.

Concluding remarks.

The design of ARCA suggests a number of directions for further work. As discussed in [1] and [2], definitive notations may be useful in other interactive applications, not necessarily concerned with graphics. In the context of notations for graphics, the use of Cayley's method of vertex referencing may also find wider applications.

A system to display and manipulate CDs might appear to be only of educational or recreational interest, but the possibility of applications cannot be dismissed. CDs implicitly occur wherever there is symmetry, and their layout could be relevant (for instance) to certain problems of circuit design.

Acknowledgements.

I am grateful to Robin Milner for pointing out the "definitive" (i.e. definition-based) nature of ARCA, and his valuable suggestions for simplifying the original form of the notation as described in [1]. I am also indebted to Kevin Murray for developing an ARCA interpreter.

References.

[1] W.M.Beynon

A definition of the ARCA notation, TC report No.54, The University of Warwick, 1983.

[2] W.M.Beynon

Definitive notations for interaction, Proc. of hci'85 "People and Computers: Designing the Interface", CUP, 1985.

[3] E. W. Dijkstra

A discipline of programming, Prentice Hall, Englewood Cliffs, NJ, 1976.

[4] D.L.Johnson

Presentation of Groups, LMS Lecture Note Series 22, CUP 1976.

[5] Marshall Hall

The Theory of Groups, Macmillan, 1959.

[6] H.Maschke

The representation of finite groups, especially the rotation groups of the regular bodies in three- and four-dimensional space, by Cayley's colour diagrams, Amer.J.Math. 18, 156-194, 1896.

[7] I.E.Sutherland

SKETCHPAD: A Man-Machine Graphical Communication System, TR No.296, Lincoln Laboratory, MIT, 1963.

[8] S.C.Johnson

YACC: Yet Another Compiler-Compiler, CSTR No.32, Bell Labs., Murray Hill, New Jersey, 1975.

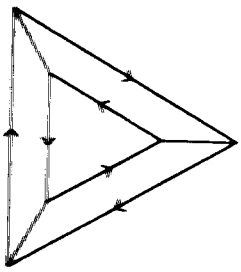

Fig . 1

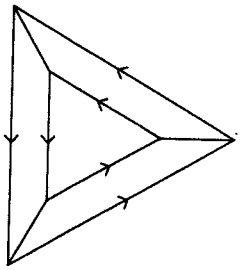

Fig . 2

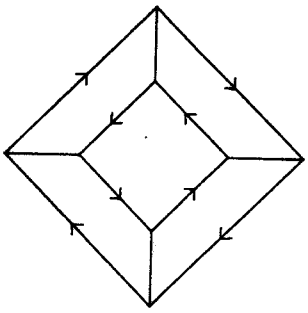

Fig . 3

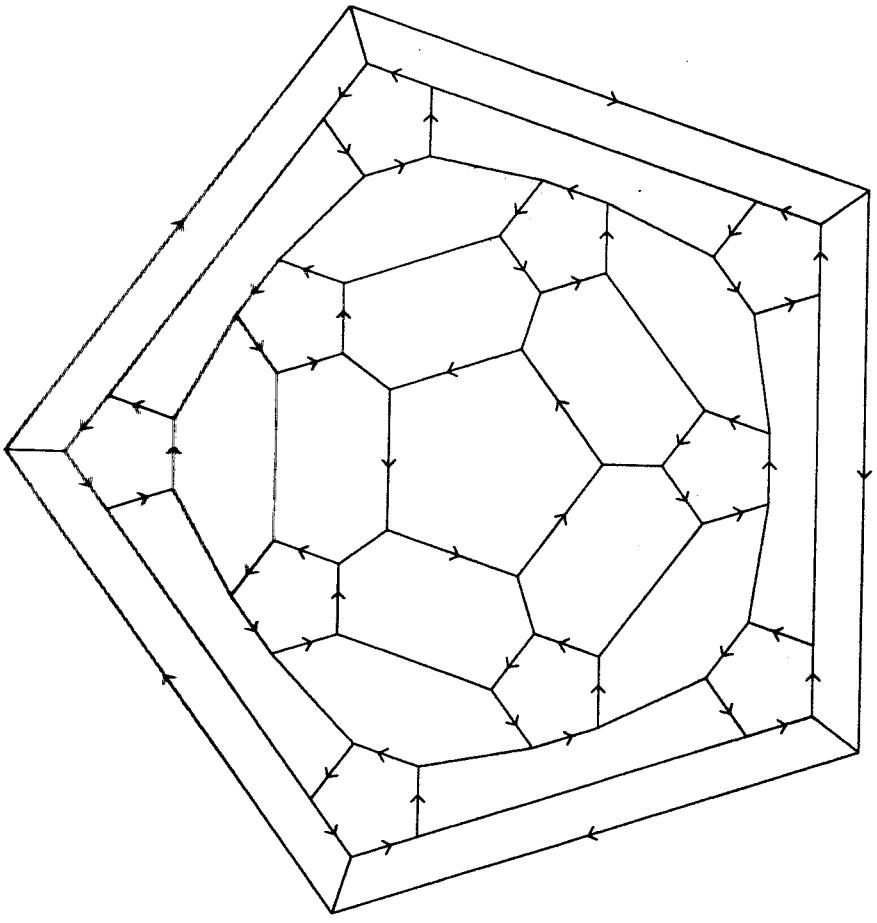

Fig . 4